Names, Scopes, and Bindings

- **Names** are identifiers (mnemonic character strings used to represent something in the program - instead of low-level concepts like addresses):
  - A name can also represent an *abstraction* of a complicated program fragment (e.g., name of a method (*control abstraction*), class (*data abstraction*), module).
  - Some symbols (like '+' ) can also be names
  - *A binding* is an association between two things, such as a name and the thing it names
  - In general, *binding time* refers to the notion of resolving any design decision in a language implementation (e.g., an example of a static binding is a function call: the function referenced by the identifier cannot change at runtime)
The textual region of the program in which a binding is active is its **scope**.

The complete set of bindings in effect at a given point in a program is known as the current **referencing environment**.
Bindings

- **Binding Time** is the point at which a binding is created or, more generally, the point at which any implementation decision is made.

- There are many times when decision about the binding are taken:
  - **language design time**: the control flow constructs, the set of fundamental (primitive) types, the available constructors for creating complex types, and many other aspects of language semantics are chosen when the language is designed.
  - **language implementation time**: precision (number of bits) of the fundamental types, the coupling of I/O to the operating system’s notion of files, the organization and maximum sizes of stack and heap, and the handling of run-time exceptions such as arithmetic overflow.
Bindings

- **program writing time**: programmers choose algorithms and names
- **compile time**: compilers plan for data layout (the mapping of high-level constructs to machine code, including the layout of statically defined data in memory)
- **link time**: layout of whole program in memory (virtual addresses are chosen at link time), the linker chooses the overall layout of the modules with respect to one another, and resolves intermodule references
- **load time**: choice of physical addresses (the processor’s memory management hardware translates virtual addresses into physical addresses during each individual instruction at run time)
Bindings

- Run time is a very broad term that covers the entire span from the beginning to the end of execution:
  - program start-up time
  - module entry time
  - elaboration time (point a which a declaration is first "seen")
  - procedure entry time
  - block entry time
  - statement execution time

- The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time
In general, later binding times are associated with greater flexibility.

Early binding times are associated with greater efficiency.

Compiled languages tend to have early binding times.

Interpreted languages tend to have later binding times.

Some languages try to do both (e.g., Java JVM).
Lifetime and Storage Management

- Bindings key events:
  - creation of objects
  - creation of bindings
  - references to variables (which use bindings)
  - (temporary) deactivation of bindings
  - reactivation of bindings
  - destruction of bindings
  - destruction of objects
The period of time between the creation and the destruction of a *name-to-object binding* is called the binding’s *lifetime*:

- If object outlives binding it's *garbage*
- If binding outlives object it's a *dangling reference*, e.g., if an object created via the C++ `new` operator is passed as a `&` parameter and then deallocated (`delete`-ed) before the subroutine returns
Lifetime and Storage Management

- **Storage Allocation** mechanisms are used to manage the object’s space:
  - **Static**: the objects are given an absolute address that is retained throughout the program’s execution
  - **Stack**: the objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.
  - **Heap**: the objects may be allocated and deallocated at arbitrary times (require a complex storage management mechanism).
Lifetime and Storage Management

- **Static allocation** for:
  - code (and small constants - often stored within the instruction itself)
  - globals
  - static or own variables
  - explicit constants (including strings, sets, etc.), e.g., printf("hello, world\n") (called *manifest constants* or *compile-time constants*, e.g., *interned strings*).

- **Arguments and return values**: Fortran (Fortran didn't have subroutine recursion) and Basic (Basic didn't have function-level scopes).
- **Temporaries** (intermediate values produced in complex calculations)
- **Bookkeeping information** (subroutine’s return address, a reference to the stack frame of the caller (the *dynamic link*), additional saved registers, debugging information)
Lifetime and Storage Management

- **Stack:**
  - Why a **stack**?
    - allocate space for recursive **routines**
    - reuse space
  - Each instance of a subroutine at run time has its own **frame** (or **activation record**) for:
    - parameters
    - local variables
    - temporaries (return address)
Lifetime and Storage Management

- Maintenance of the stack is the responsibility of the subroutine calling sequence (the code executed by the caller immediately before and after the call), which includes: the prologue (code executed at the beginning) and epilogue (code executed at the end) of the subroutine itself.
Lifetime and Storage Management

- **Stack pointers:**
  - The *frame pointer* \((fp)\) register points to a known location within the frame of the current subroutine
    - \(fp\) usually points to the parameters (above the return address) for the current call
  - The *stack pointer* \((sp)\) register points to the first unused location on the stack (or the last used location on some machines)
    - \(sp\) would point to where arguments would be for next call
    - Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time
Lifetime and Storage Management

Direction of stack growth (usually lower addresses)

Subroutine A
Subroutine B
Subroutine C
Subroutine D

Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address

fp (when subroutine C is running)

sp

procedure C
D; E
procedure B
if ... then B else C
procedure A
B
-- main program
A

(c) Paul Fodor (CS Stony Brook) and Elsevier
Calling Methods Example in Java

(a) The main method is invoked.

```
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);

    System.out.println("The maximum between " + i + " and " + i + " is " + k);
}
```

(b) The max method is invoked.

```
public static int max(int num1, int num2) {
    int result:
    if (num1 > num2)
        result = num1;
    else
        result = num2:

    return result:
```
Trace Call Stack

public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);

    System.out.println("The maximum between " + i + " and " + j + " is " + k);
}

public static int max(int num1, int num2) {
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
public static void main(String[] args) {  
int i = 5;  
int j = 2;  
int k = max(i, i);  
System.out.println(
  "The maximum between " + i + " and " + i + " is " + k);  
}  

public static int max(int num1, int num2) {  
  int result:  
  if (num1 > num2)  
    result = num1:  
  else  
    result = num2:  
  return result:  
}
Trace Call Stack

The main method is invoked.
Space required for the main method

```java
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);
    System.out.println("The maximum between " + i + " and " + i + " is " + k);
}
```

```java
public static int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```
Trace Call Stack

public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, i);
    
    System.out.println("The maximum between " + i + ", " + j + " is " + k);
}

public static int max(int num1, int num2) {
    int result;
    
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    
    return result;
}
Trace Call Stack

```java
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j):

    System.out.println(
        "The maximum between " + i + 
        " and " + j + " is " + k);
}
```

```java
public static int max(int num1, int num2) {
    int result:

    if (num1 > num2)
        result = num1:
    else
        result = num2:

    return result:
}
```

pass the values of i and j to num1 and num2

num2: 2
num1: 5

Space required for the main method

k:
j: 2
i: 5

The max method is invoked.
Trace Call Stack

```java
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, i):

    System.out.println(
        "The maximum between " + i + " and " + j + " is " + k);
}

public static int max(int num1, int num2) {
    int result:
    if (num1 > num2)
        result = num1:
    else
        result = num2:
    return result:
}
```

pass the values of i and j to num1 and num2

result:
num2: 2
num1: 5

Space required for the main method
k: 5
j: 2
i: 5

The max method is invoked.
Trace Call Stack

```
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);

    System.out.println(
        "The maximum between " + i + 
        " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
    int result:

    if (num1 > num2)
        result = num1:
    else
        result = num2:

    return result:
}
```

(num1 > num2) is true

Space required for the main method
k: 
j: 2
i: 5

The max method is invoked.

result:
num2: 2
num1: 5

(c) Paul Fodor (CS Stony Brook) and Elsevier
Trace Call Stack

public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j):

    System.out.println("The maximum between " + i + " and " + j + " is " + k);
}

public static int max(int num1, int num2) {
    int result:

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result:
}
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);

    System.out.println("The maximum between " + i + " and " + j + " is " + k);
}

public static int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
## Trace Call Stack

### The main method is invoked.

```java
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);

    System.out.println("The maximum between "+ i + " and "+ j + " is "+ k);
}
```

### Space required for the main method
- k: 5
- j: 2
- i: 5

```
public static int max(int num1, int num2) {
    int result:

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```
Lifetime and Storage Management

• Reminder: maintenance of stack is responsibility of *calling sequence* and subroutines *prolog* and *epilog*

• Optimizations:
  • space is saved by putting as much in the prolog and epilog as possible
  • time may be saved by
    • putting stuff in the caller instead or
    • combining what's known in both places (interprocedural optimization)
  • Unfolding subroutines (e.g., Prolog/Datalog unfolding optimizations)

• One cannot return references to objects on the stack
  • Rookie mistake in C: the lifetime is limited to function scope.
Lifetime and Storage Management

- Heap-Based Allocation
  - **Heap** is for **dynamic** allocation
  - A heap is a region of storage in which sub-blocks can be allocated and deallocated at arbitrary times
  - dynamically allocated pieces of data structures: objects, Strings, lists, and sets, whose size may change as a result of an assignment statement or other update operation
Lifetime and Storage Management

- **Fragmentation:**
  - *Internal fragmentation* occurs when a storage-management algorithm allocates a block that is larger than required to hold a given object.
  - *External fragmentation* occurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, unused space is composed of multiple blocks: there may be quite a lot of free space, but no one piece of it may be large enough to satisfy some request.
The storage-management algorithm maintains a single linked list, the free list, of heap blocks not currently in use.

- The first fit algorithm selects the first block on the list that is large enough to satisfy a request.
- The best fit algorithm searches the entire list to find the smallest block that is large enough to satisfy the request.

Common mechanisms for dynamic pool adjustment:
- The buddy system: the standard block sizes are powers of two.
- The Fibonacci heap: the standard block sizes are the Fibonacci numbers.

- Compacting the heap moves already-allocated blocks to free large blocks of space.
Garbage Collection (GC):

- In languages that deallocation of objects is not explicit
  - Manual deallocation errors are among the most common and costly bugs in real-world programs
- Objects are to be deallocated implicitly when it is no longer possible to reach them from any program variable
  - Costly
- Methodologies: reference counting, Mark/Sweep, Copying, Generational GC
  - Stop-the-world vs. incremental vs. concurrent
- Generational GC (Java): hierarchy of memory sectors, "young" vars (often deleted after 1 reference) and "old"
Scope Rules

- The binding *scope* is the textual region of the program in which a binding is active
  - A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted
- The scope of a binding is determined *statically* or *dynamically*
- Scoping rule example 1: Declaration before use
  - Can a name be used before it is declared?
    - Java local vars: NO
    - Java class properties and methods: YES
Scope Rules

- Scoping rule example 2:
  - Two uses of a given name
    - Do they refer to the same binding?
      
      ```python
      a = 1
      ...
      def f():
        a = 2
        b = a
      ```

  - the scoping rules determine the scope
In static scope rules, bindings are defined by the physical (lexical) structure of the program.

**Static scoping** (also called lexical scoping) rule examples:

- one big scope (old Basic),
- scope of a function (variables live through a function execution)
- block scope (a local var. is available in the block in which is defined)
- nested subroutines (have access to the variables defined in the parent)
- if a variable is active in one or more scopes, then the closest nested scope rule applies

Lexical/static scoping was used for ALGOL and has been picked up in most other languages since then: like Pascal, C, Java.
Scope Rules

- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for new local variables,
  - deactivate bindings for global variables that are re-declared (these variables are said to have a "hole" in their scope), and
  - make references to reference parameters.

- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated
Scope Rules

• **ELABORATION** = process of creating bindings when entering a subroutine scope:
  • storage may be allocated
  • even exceptions propagated as a result of the elaboration of declarations
Static Scoping

• With **STATIC (LEXICAL) SCOPE RULES** (e.g., C, Java and python), a scope is defined in terms of the physical (lexical) structure of the program:
  • The determination of scopes can be made by the compiler
  • All bindings for identifiers can be resolved by examining the program
  • Typically, we choose the most recent, active binding made at compile time
    • Most compiled languages, C, Pascal, Java and python included, employ static scope rules
Static Scoping

- **Nested blocks:**
  - The classical example of static scope rules is the *most closely nested rule* used in block structured languages (started with Pascal):
    - An identifier is known in the scope in which it is declared and in each enclosed scope, unless it is re-declared in an enclosed scope (the original identifier is said to be hidden)
    - To resolve a reference to an identifier, we examine the local scope and statically enclosing scopes until a binding is found
  - **Classes** (in object-oriented languages) have even more sophisticated (static) scope rules
procedure P1(A1 : T1);
var X : real;
...
    procedure P2(A2 : T2);
    ...
        procedure P3(A3 : T3);
        ...
            begin
            ...
                (* body of P3 *)
            end;
            ...
            begin
            ...
                (* body of P2 *)
            end;
            ...
            procedure P4(A4 : T4);
            ...
                function F1(A5 : T5) : T6;
                var X : integer;
                ...
                begin
                ...
                    (* body of F1 *)
                end;
                ...
                begin
                ...
                    (* body of P4 *)
                end;
                ...
                begin
                ...
                    (* body of P1 *)
                end;
end
Static Scoping

- The simplest way in which to find the frames of surrounding scopes is to maintain a static link in each frame that points to the “parent” frame (i.e., the frame of the most recent invocation of the lexically surrounding subroutine).

- If a subroutine is declared at the outermost nesting level of the program, then its frame will have a null static link at run time.

- If a subroutine is nested k levels deep, then its frame’s static link, and those of its parent, grandparent, and so on, will form a static chain of length k at run time.
Nesting of subroutines:  During run time (with links)
Static Scoping

- **Declaration Order rules:**
  - Several early languages, including Algol 60 and Lisp, required that all declarations appear at the beginning of their scope.
  - Pascal modified the requirement to say that names must just be declared before they are used.
  - However, it may still use a whole-block scope (implemented in C#)
    - Example in C#: defining a variable in a block makes every external declaration hidden, so the M=N assignment generates a compiling error:

```csharp
class A {
    const int N = 10;
    void foo() {
        const int M = N;    // uses inner N before it is declared
        const int N = 20;
    }
}```
Dynamic Scoping

- **Dynamic scope rules**: bindings depend on the current state of program execution:
  - They cannot always be resolved by examining the program because they are dependent on calling sequences
    - The binding might depend on how a function is called
  - To resolve a reference, we use the most recent, active binding made at run time
Dynamic Scoping of bindings

- Example:

```python
var total = 0
def add():
    total += 1
def myfunc():
    var total = 0
    add()
    add()
myfunc()
print total
```

The stack trace shows:
- `main total=1`
- `add` (called from `main`)
- `myfunc total=1` (called from `add`)

This prints 1
User input in binding decisions

Program scopes (input, output)

```plaintext
program scopes (input, output)

var a : integer;
procedure first();
  begin a := 1; end;
procedure second();
  var a : integer;
  begin first(); end;
begin
  a := 2;
  if read_integer() > 0
    second();
  else
    first();
  write(a);
end.
```

- Program output depends on both scope rules and, in the case of dynamic scoping, a value read at run time
  - If static scoping is in effect, this program prints a 1 (the global a).
  - If dynamic scoping is in effect, the output depends on the value read at line 10 at run time: if the input is positive, the program prints a 2; otherwise it prints a 1.
- The assignment to the variable a at line 4 refers either to the global variable declared at line 2 or to the local variable declared at line 6.
Dynamic Scoping of bindings

- **Dynamic scope rules** are usually encountered in interpreted languages.
  - **Lisp, Perl, Ruby, C#, the shell languages bash, dash, and PowerShell**
    - Such languages do not always have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect.
- A common use of dynamic scope rules is to provide implicit parameters to subroutines.
  - Alternative mechanisms exist
    - default and optional parameters
Binding of Reference Environments

• A reference environment = all the bindings active at a given time.

• When we take a reference to a function, we need to decide which reference environment we want to use!!!

• Deep binding binds the environment at the time the procedure is passed as an argument

• Shallow binding binds the environment at the time the procedure is actually called
Example: static vs. dynamic shallow vs. dynamic deep

```python
var total = 0
def a():
    total += 1
def b(F):
    var total = 0
    F()
    print "B", total
def c():
    var total = 0
    b(a)
    print "C", total
c()
print "T", total
```

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic shallow</th>
<th>dynamic deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: static vs. dynamic shallow vs. dynamic deep

```python
def a():
    total += 1  (* the state of the stack *)

def b(F):
    var total = 0
    F()
    print "B", total

def c():
    var total = 0
    b(a)
    print "C", total

c()
print "T", total
```

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic shallow</th>
<th>dynamic deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Stack

```
main       total=1
            c           total=0
            b           F = a
            a
```

(c) Paul Fodor (CS Stony Brook) and Elsevier
Example: static vs. dynamic shallow vs. dynamic deep

```python
var total = 0
def a():
    total += 1 (* the state of the stack)
def b(F):
    var total = 0
    F()
    print "B", total
def c():
    var total = 0
    b(a)
    print "C", total
c()
print "T", total
```

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic shallow</th>
<th>dynamic deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Stack

```
main     total=0
    c           total=0
        total=1
    b        F = a
        total=0
    a
main     total=0
```
Example: static vs. dynamic shallow vs. dynamic deep

```python
var total = 0
def a():
    total += 1  # (* the state of the stack)
def b(F):
    var total = 0
    F()
    print "B", total
def c():
    var total = 0
    b(a)
    print "C", total
c()
p
```
x: integer := 1
y: integer := 2

procedure add
    x := x + y

procedure second(P:procedure)
    x:integer := 2
    P()

procedure first
    y:integer := 3
    second(add)

first()
write_integer(x)

- **static scoping** would print 3.

- **shallow binding** just traverses up until it finds the nearest variable that corresponds to the name (increments the local x), so the answer would be 1.

- **dynamic scoping with deep binding**: when add is passed into second the environment of add is $x = 1, y = 3$ and the x is the global x so it writes 4 into the global x, which is the one picked up by the write_integer.
x: integer := 1
y: integer := 2

procedure add
  x := x + y

procedure second(P: procedure)
  x: integer := 2
  P()

procedure first
  y: integer := 3
  second(add)

first()
write_integer(x)

- static scoping would print 3.
Binding of Reference Environments

- shallow binding just traverses up until it finds the nearest variable that corresponds to the name (increments the local x), so the answer would be 1.

```plaintext
x: integer := 1
y: integer := 2

procedure add
  x := x + y

procedure second(P:procedure)
  x: integer := 2
  P()

procedure first
  y: integer := 3
  second(add)

first()
write_integer(x)
```

Stack

- add
  - x = 2
- second P = add
- first y = 3
- main y = 2
  - x = 1
**Binding of Reference Environments**

- *dynamic scoping with deep binding*: when `add` is passed into `second` the environment of `add` is `x = 1, y = 3` and the `x` is the global `x` so it writes 4 into the global `x`, which is the one picked up by the `write_integer`.

```plaintext
x: integer := 1
y: integer := 2

procedure add
  x := x + y

procedure second(P:procedure)
  x:integer := 2
  P()

procedure first
  y:integer := 3
  second(add)

first()
write_integer(x)
```

```plaintext
P=add
  x=2
second P = add
  first y=3
  main y=2
    x=1
```

```plaintext
P=add
  x=5
second P = add
  first y=3
  main y=2
    x=4
```
Closure

- **Deep binding** is implemented by creating an explicit representation of a referencing environment and bundling it together with a reference to the subroutine (this bundle is called *closure*).

```python
def foo():
    a = 100
    def bar():
        return a
    a += 1
    return bar
f = foo()
print f()
```

Outputs: 101

Implementing closures means we may need to keep around `foo`'s frame even after `foo` quits.

- heap allocation to keep around objects pointed to by `foo`.
- In languages without closures, we can fake them with objects.
Binding of Referencing Environments

- Accessing variables with dynamic scope:
  - (1) keep a stack (association list) of all active variables
    - When you need to find a variable, hunt down from top of stack
    - This is equivalent to searching the activation records on the dynamic chain
  - (2) keep a central table with one slot for every variable name
    - You'll need a hash function or something to do lookup
    - Every subroutine changes the table entries for its locals at entry and exit.
      - In effect, variable lookup in a dynamically-scoped language corresponds to symbol table lookup in a statically-scoped language
The Meaning of Names within a Scope

- Overloading:
  - same name, more than one meaning
  - some overloading happens in almost all languages
    - integer + vs. real + vs. String concatenation
    - read and write in Pascal are overloaded based on the number and types of parameters
    - function return in Pascal
  - some languages get into overloading in a big way: Java, C++
public class AmbiguousOverloading {
    public static void main(String[] args) {
        System.out.println(max(1, 2));
    }
    public static double max(int num1, double num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
    public static double max(double num1, int num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
}
The Meaning of Names within a Scope

- It's worth distinguishing between some closely related concepts:
  - overloaded functions - two different things with the same name
    - overload `norm`
      int norm(int a){return a>0 ? a : -a;)
    complex norm(complex c ) { ... }
  - polymorphic functions: one thing that works in more than one way
    - Overriding in OO programming, and
    - Generic programming:
      function min (A : array of Comparable)
      - generic functions - a syntactic template that can be instantiated in more than one way at compile or even run time
        - via macro processors in C++
Polymorphism, Dynamic Binding and Generic Programming

```java
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}
class GraduateStudent extends Student {
}
class Student extends Person {
    public String toString() {
        return "Student";
    }
}
class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

Method `m` takes a parameter of the `Object` type – can be invoked with any object.

**Polymorphism**: an object of a subtype can be used wherever its supertype value is required.

**Dynamic binding**: the Java Virtual Machine determines dynamically at runtime which implementation is used by the method.

When the method `m(Object x)` is executed, the argument `x`’s `toString` method is invoked.
Dynamic Binding

• Suppose an object o is an instance of classes $C_1$, $C_2$, ..., $C_{n-1}$, and $C_n$
  • $C_1$ is a subclass of $C_2$, $C_2$ is a subclass of $C_3$, ..., and $C_{n-1}$ is a subclass of $C_n$
  • $C_n$ is the most general class, and $C_1$ is the most specific class
• If o invokes a method p, the JVM searches the implementation for the method p in $C_1$, $C_2$, ..., $C_{n-1}$ and $C_n$, in this order, until it is found, the search stops and the first-found implementation is invoked

Since o is an instance of $C_1$, o is also an instance of $C_2$, $C_3$, ..., $C_{n-1}$, and $C_n
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}
class Student extends Person {
    public String toString() {
        return "Student";
    }
}
class Person extends Object {
    public String toString() {
        return "Person";
    }
}
Method Matching vs. Binding

• The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.

• The Java Virtual Machine dynamically binds the implementation of the method at runtime.
The Meaning of Names within a Scope

- **Aliasing**: two names point to the same object
  - Makes program hard to understand
  - Makes program slow to compile
  - What are aliases good for?
    - linked data structures
  - Example: passing a variable by reference to a subroutine that also accesses that variable directly

```c
double sum, sum_of_squares;
accumulate(sum);
void accumulate(double& x){// x is passed by reference
    sum += x;
    sum_of_squares += x * x;
}
```

sum is passed as an argument to accumulate, so, sum and x will be aliases for one another
Modules (AKA packages)

- Break program up into parts, which need to be explicitly imported.
- We only need to agree on the meaning of names when our code interacts.
Macros

- Macros are a way of assigning a name to some syntax.
  - C: Textual substitution.

\[
\text{\#define MAX(x, y) \((x > y ? x : y)\)}
\]

  - benefit: shorter code, no stack, can choose not to execute some of the code

- Problems with macros:
  - multiple side effects: \texttt{MAX(a++, b++)}
  - scope capture: temporary \texttt{var} used inside macro has same name as a real \texttt{var} — for example: \texttt{t} exists outside

\[
\text{\#define SWAP(a,b) \{t = (a); (a) = (b); (b) = t;\}}
\]

  - Scheme and Common Lisp hygienic macros rename variables